Guide to the Data Acquisition Engine

Fermilab
Beams Division
Accelerator Controls Department

Kevin Cahill, Charlie King

created
April 17, 2001

Version 1.0.3
May 21, 2002

Architecture

Client Server Model

Clients are end-user applications existing here at Fermilab, in a home, or at any workstation connected to the Internet. Servers are control system applications and nodes that serve clients and are privileged to speak the proprietary ACNET protocol and utilize resources of the control system for itself and on behalf of authorized clients. A server is referred to as Data Acquisition Engine, DAE, or engine. The connection between the client and engine is RMI, Remote Method Invocation, a portable networking protocol between Java applications. Consequently, many client applications are written in Java. Non-Java clients are also supported by engines using the ACNET or XML-RPC protocols. XML-RPC servers are engine clients submitting data acquisition jobs on behalf of clients written in any language that supports XML-RPC.

Server nodes reside on high-bandwidth networks, emit and listen to multicast networking messages, work with raw byte-ordered data, and work together to consolidate traffic and functions across the control system.

Client nodes may be located across firewall routers, may exist on networks where multicasting is blocked, will normally work with scaled data, and rely on a server connection to the control system to acquire accelerator data.

Development

Development tools are unrestricted, but an integrated development environment, IDE, improves productivity. Most development occurs on Windows platforms, and the integrated development environments JBuilder, VisualCafe, and Forte for Java have all been used to develop client and server code. JBuilder is the currently recommended choice for an IDE.

Security

User authorization is required. A frequent user may persistently register a node the user solely controls. Other users will be required to login as each first connection from their node to a server is made. The login will employ kerberos. Each user, application, and node in the system may be assigned a set of permissions. Permission to set a device requires the binary AND of all permissions to result in TRUE.

Accountability

Connections, jobs, bandwidth, device setting and control, and other client characteristics will be logged for understanding and as a necessary security element.

Callbacks

Asynchronous callbacks are the rule, and polling is generally not employed.

Thin clients

An important consideration for building reusable controls' components is to provide for execution of the controls logic on the server when possible. The VAX/VMS control system promoted fat clients in that server applications simply acquired data, and user applications contained the controls' logic. A data logger's logic for logging and retrieving data is more reusable when developed as server logic instead of client logic.

Data acquisition job

Overview

This accelerator control system exists primarily to display and control accelerator components and systems. The data acquisition job is used to acquire and set data. The job defines the 'from', 'to', 'what', 'when', 'who', and 'how' objects of data acquisition.

Job components

Source

The source component of the job determines where the data comes **from.** AcceleratorSource, SavedDataSource, and DataLoggerSource are examples of . DataSource, the abstract class all data sources data extend.

Disposition

The disposition component of the job determines where the data is delivered **to.** AcceleratorDisposition, SavedDataDisposition, and DataLoggerDisposition are examples of DataSource, the abstract class all data dispositions extend. The job components for source and disposition extend the same object since source and disposition are often interchangeable.

Item

The item component of the job determines **what** data is collected from the source and delivered to the disposition. AcceleratorDevicesItem, ColliderShotItem, and ParameterPageItem are examples of DataItem, the abstract class all data items extend.

Event

The event component of the job determines **when** or how often data is collected or returned. OnceImmediateEvent, DeltaTimeEvent, and ClockEvent are examples of DataEvent, the abstract class all data event extend.

User

The user component of the job specifies **who** owns the job and its connection to the server. The security and accountability issues are encompassed by the DaqUser job component.

Control

The control component of the job determines **how** the job recovers from lost connections, and DaqJobControl is a job control object. DaqJobControl also provides for callback to the interfaces DaqJobCompletion, DaqJobTroubles, and DaqJobStatistics.

Code Example, one-shot read

```
import gov.fnal.controls.daq.acquire.*;
import gov.fnal.controls.daq.datasource.*;
import gov.fnal.controls.daq.items.*;
import gov.fnal.controls.daq.events.*;

(within some class)

DaqUser user = new DaqUser ("ReadingTry", "DSE03.fnal.gov") ;
int property = AcceleratorObject.READING;
AcceleratorDevice device = new AcceleratorDevice("M:OUTTMP", property,
0,0);

DataSource from = new AcceleratorSource();
DataSource to = new DaqEngineTerminalOutputDisposition("Testing",
true);
AcceleratorDevicesItem item = new AcceleratorDevicesItem();
item.addDevice(device);
DataEvent event = new OnceImmediateEvent();
DaqJobControl control = new DaqJobControl();

DaqJob job = new DaqJob(from, to, item, event, user,control);

try
{
	job.start();
}
catch (Exception e)
{
	System.out.println("whoops, job.start caught: " + e);
}
```

Explanation

The first 4 lines import packages supporting data acquisition that are needed for the code that follows.

The next line establishes a connection with a server engine, in this case the default engine since it is not specified. The user may be prompted to log in so the engine can determine the privileges of this user. Most applications need but one DaqUser connection for all the DaqJob (s) of their application.

The next line identifies the data acquisition property of interest.

The next line defines the object describing the outdoor temperature reading.

The next line specifies the data source is the accelerator, i.e. collect data from the front-ends in real time.

The next line specifies the disposition to be the system terminal output.

The next line creates a collection frequency object specifying a collection of but once.

The next line creates a default job control object.

The next line creates the job.

The next line tries to start the job. Catching and printing the exception to the terminal will describe the problem starting the job.

This job will stop by itself once data is returned.

RMI

As the job is started, the job components (excepting job control) are sent to a server engine via Remote Method Invocation (RMI). RMI is a portable networking protocol allowing the engine to communicate with clients regardless of their byte architecture. The engine examines the job components to determine which kind of job scheduler to start to carry out the job.

Disposition Callbacks

GenericCallback

Consider an application collecting readings where the job source is AcceleratorSource, item is an AcceleratorDevicesItem containing an AcceleratorDevice, and event is a DeltaTimeEvent. By definition, the job's disposition must implement the GenericCallback interface (that is how the engine knows to start an AcceleratorPoolScheduler to return readings). The class that started the job or an inner class might have extended MonitorChangeDisposition and overloaded the reading method. Failing to overload the reading method would result in the reading method of MonitorChangeDisposition receiving the reading and writing a message describing the reading to System.out. The callback signature is:

public void reading(WhatDaq device, int element, int error, Date timestamp, CollectionContext context, double reading)

and not difficult to implement and understand. The application is not limited to receiving all or any of the readings within the disposition's callback. When the item or device in the job implements GenericCallback, the callback order of delivery preference is device, item, and disposition.

PlotCallback

An application obtaining plot data, specifies the PlotCallback implementer in the PlotDataItem. The callback signature is:

public void plotData(Object request, Date timestamp, CollectionContext context, int error, int numberPoints, long[] microSecs, short[] nanoSecs, double[] values)

EventCallback

An application requesting Tevatron clock event or software state transition event callbacks specifies the DataEventObserver in the EventDataItem. The callback signature is:

public void update(DataEvent request, DataEvent reply)

ReportCallback

An application obtaining report callbacks specifies the ReportCallback implementer in the ReportItem or disposition. The callback signatures are:

public void initialReport(String report, Date reportTime)
public void updateReport(String report, Date reportTime)
public void troubleReport(String report, Date reportTime)
public void finalReport(String report, Date reportTime)

Other Callbacks

Other callbacks exist to return objects, trees, and ACNET errors. Their callbacks include ObjectCallbackDisposition, FermiDataDispositionWithCallback, TreeCallback, and AcnetErrorCallback. The callback signatures are designed to be easy to understand and implement.

Database Access

The package gov.fnal.controls.db contains support for database access. Pooled and transparent database connections are supported. When possible, job element database access occurs on the engine to reduce the job's RMI overhead and to promote performance.

ACNET errors and exceptions

AcnetError in the package acnet supports the definition, translation, caching, and report generation of ACNET facility code, error code status returns.

AcnetException and AcnetErrorException in the package util support the creation of exceptions that are control system specific.

Schedulers

AcceleratorPoolScheduler, DataLoggerScheduler, and SnapShotScheduler are examples of schedulers. The scheduler is responsible for starting and completing the job on the engine. In the outdoor temperature example, an AcceleratorPoolScheduler inquires of the AcceleratorDevicesItem what devices are to be collected, inserts those devices into the data pools appropriate for the OnceImmediateEvent, informs the disposition how to determine when the job is complete, notifies each of the job components that the job is beginning, notifies each of the job components as the job ends, and returns completion status back to the job within the client. The scheduler returns a DataSchedulerInterface to the client's job, an RMI connection that allows the client to cancel the job.

Interchangeable job components

The DaqJob and its components are intended to promote reusability. Consider the outdoor temperature collection job. Substituting a DeltaTimeEvent or ClockEvent for the OnceImmediateEvent and the disposition to a PlotDisposition changes the role of the job to a plotting application. Substituting a SavedDataSource for AcceleratorSource displays a value from the past. Substituting a DataLoggerDisposition for DaqTerminalOutputDisposition logs the value.

Client applications should not expect their job components to be fixed. Consider the parameter page application. The parameter page application is implemented as a disposition. The normal behavior of a parameter page application is demonstrated with a job including AcceleratorSource, ParameterPageItem, and DefaultDataEvent. Job component substitution of SavedDataSource for AcceleratorSource allows the ParameterPageDisposition to be reused as a Save/Restore application.

The DaqJob is not limited to the collection of accelerator device data. The DaqJob may also be configured to inform a client application of Tevatron clock events and software state transitions. It is also used to return progress reports for a Save/Restore save and to report front-end heartbeat transitions.

Data Acquisition Engine

ACNET

The ACNET protocol is a connectionless, peer-to-peer, proprietary Fermilab networking protocol. The data acquisition, plot, and alarm message protocols are built atop ACNET messages, and ACNET is the messaging protocol supported by the front-ends.

The ACNET implementation in Java supports multiple collections, is asynchronous, supports timeouts on single and multiple replies, supports large message sizes, supports offsets into a transmitted message buffer and includes support for building and parsing a message buffer. Each engine has about 20 connected tasks supporting or using type-coded messaging protocols built upon ACNET. Client nodes, by definition, do not support ACNET.

Clients

The Data Acquisition Engine is a server supporting clients. It is the portal for client access to the control system. Clients are connected to engines with RMI. Java applets and applications are clients.

Applets

Java control system applets are essentially control system applications running with greater security restrictions concerning the RMI connection between the client and engine. When users want access to the control system from offsite, neither the user nor server wants to open its machine to unrestricted network access. Consequently, offsite access will be gained from a web server with links to control system applets. The applet is restricted to a network connection only to that web server which is connected to the control system.

Applications

Java control system applications are run from typical batch command files, start up faster than applets, and have fewer restrictions when run on site.

Internal threads

Functions such as data logging behave like client applications, but happen to be running in a thread within an engine. Jobs within the engine have the option to suppress RMI callbacks.

Bridge to front-ends

The engines serve as a bridge from the RMI to ACNET protocol and as a bridge from client applications to front-end applications.

Front-end to front-ends

The alarms, plotting, and data acquisition protocols are not uniformly supported across all front-end platforms. By front-ending the front-ends, the acquired consistency promotes DaqJob component reusability. Sampling on clock event and state transition plus delay, long periodic rates, as well as continuous and snapshot plot support are supported within the engine for **all** devices by operating as a front-end to front-ends.

Front-end consolidation

The VAX/VMS control system consolidates front-end data acquisition on a node basis across about 60 nodes. If each of those nodes supported a client application interested in the same device, the device's front-end would transmit about 60 messages periodically. Consolidation across all the engines reduces the front-end's responsibility to transmitting a single reply periodically to the engine servicing that front-end, and the consolidating engine must deal with sharing this data with other interested engines. The bandwidth problem moves from the front-end to the engine, but the engines are generally centrally located near high-bandwidth links and are more easily upgraded. Consolidation increases complexity and the number of layers of message handling, but is necessary if the front-ends are ever to move from the proprietary ACNET protocol to a more processor intensive, portable protocol such as RMI or CORBA.

Configuration

When a Data Acquisition Engine starts, it configures itself as a member of the operational DAE cluster, the development DAE cluster, or as a stand-alone engine.

Operational Cluster

The operational engine cluster is a group of machines that are cooperatively supporting each other, front-ends, and client nodes in delivering operational access to the control system. Some engines are assigned front-end consolidation responsibility. Engines ping each other and their assigned front-ends and consolidate front-end data acquisition across the operational cluster. Engines participate in transmitting and receiving multicast pool messages containing reading data on popular channels. Software state transitions announce the coordination control notification, and ACNET is used as the consolidation message carrier.

Development Cluster

The development engine cluster is a small group of machines with the same characteristics as the operational cluster, only utilizing independent state transition devices, multicast channels, and consolidation messaging.

Stand-alone

A stand-alone DAE is not part of a cluster. It does not participate in consolidation or otherwise share responsibility with other engines. Stand-alone engines have a direct connection to front-ends.

Other Data Acquisition Engines

Data Server Engines

A data server engine, DSE, is a data acquisition engine with no front-end consolidation responsibility primarily supporting application clients. Clients consume resources on their servers. A server starved of resources by a misbehaving client operating as a consolidation engine for some front-end reduces access to that front-end across the control system. The separation of the front-end centric DAE from the client centric DSE provides for greater reliability in the system. Data server engines often will be running on the same node as the client applications. A Main Control Room, MCR, console will run a data server engine with client applications connected to the local DSE. The DSE reaches a front-end thorough its consolidating DAE. The DSE may be shutdown or restarted, and only that MCR console is affected.

Data Utility Engines

A data utility engine, DUE, is a data acquisition engine with no front-end consolidation responsibility primarily supporting open access front-end clients, data loggers, and servers.

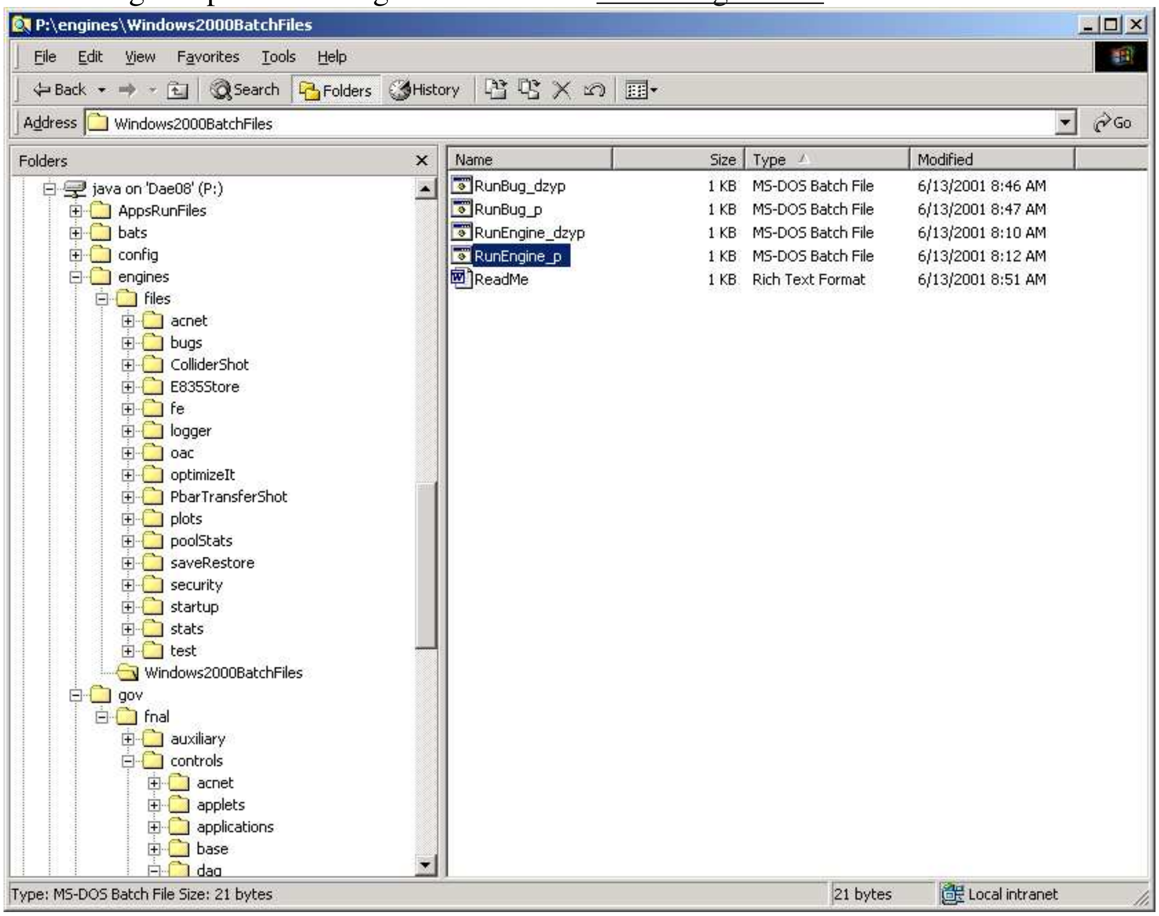
Data Programmer Engines

A data utility engine, DPE, is a data acquisition engine generally assigned to a programmer for development purposes when a desktop machine is not available or convenient.

Starting and Using an Engine

Startup command files

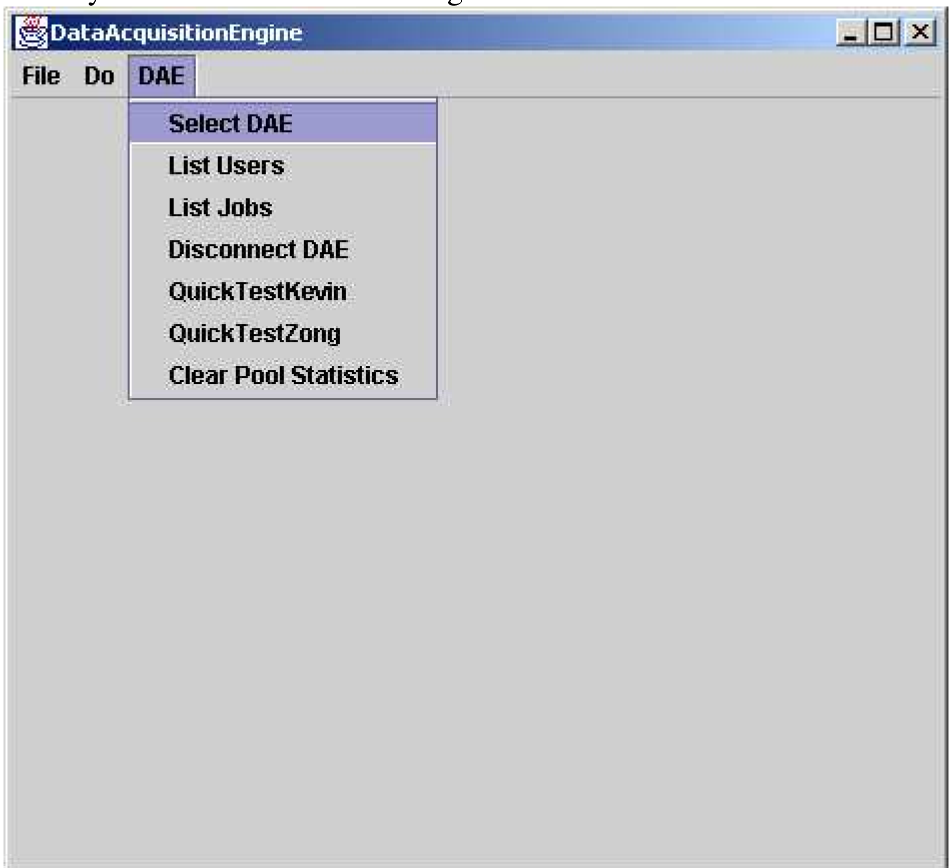
When a Data Acquisition Engine starts, it configures itself as a member of the operational DAE cluster, the development DAE cluster, or as a stand-alone engine. The instructions for starting an operational engine are found at Restarting a DAE.



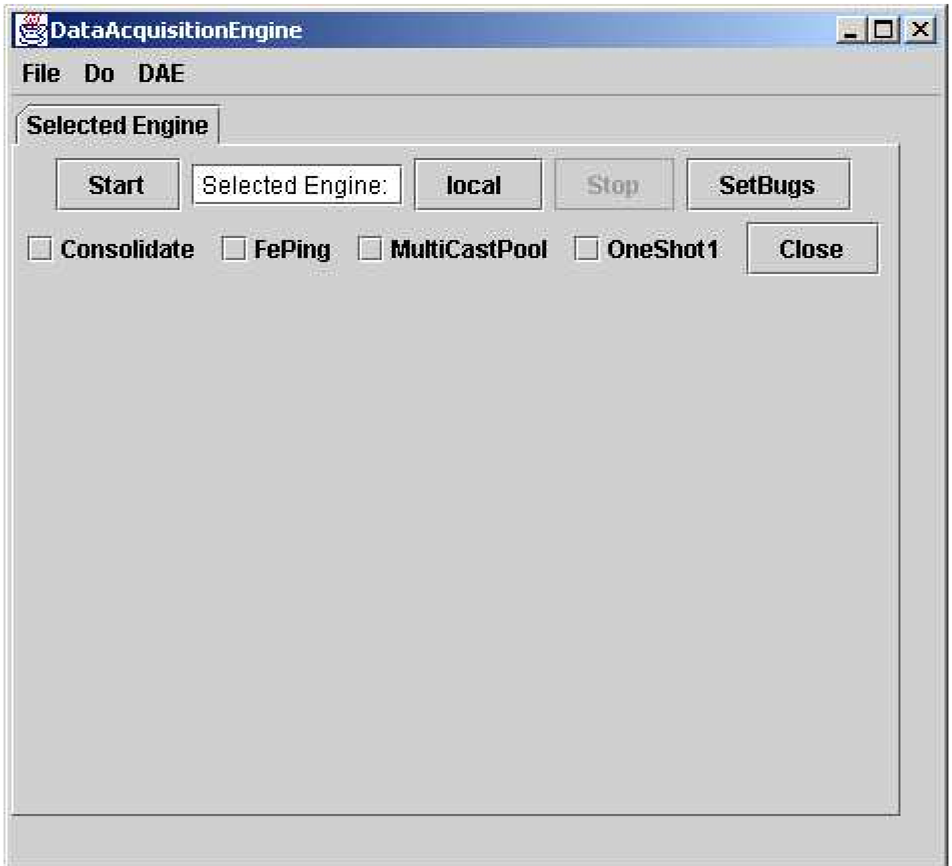
Developers may start a DAE from the path dae08:gov.fnal.controls.servers.dae. Any node with an entry in the ACNET node tables may run an engine. The batch file RunEngine_p starts a DAE on the operational engine cluster unless the node is a designated development cluster node. The batch file RunBug_p starts the DAE graphical user interface, GUI, but does not initially start ACNET and other engine facilities. The "_p" portion of the batch file implies that "p:\" is specified as the Java classpath. The batch files ending with "dzyp" have a classpath disk order of d, z, y, and p in that order.

DAEBug

The batch file RunBug_p starts the DAE graphical user interface, GUI, but does not initially start ACNET and other engine facilities.



leads to



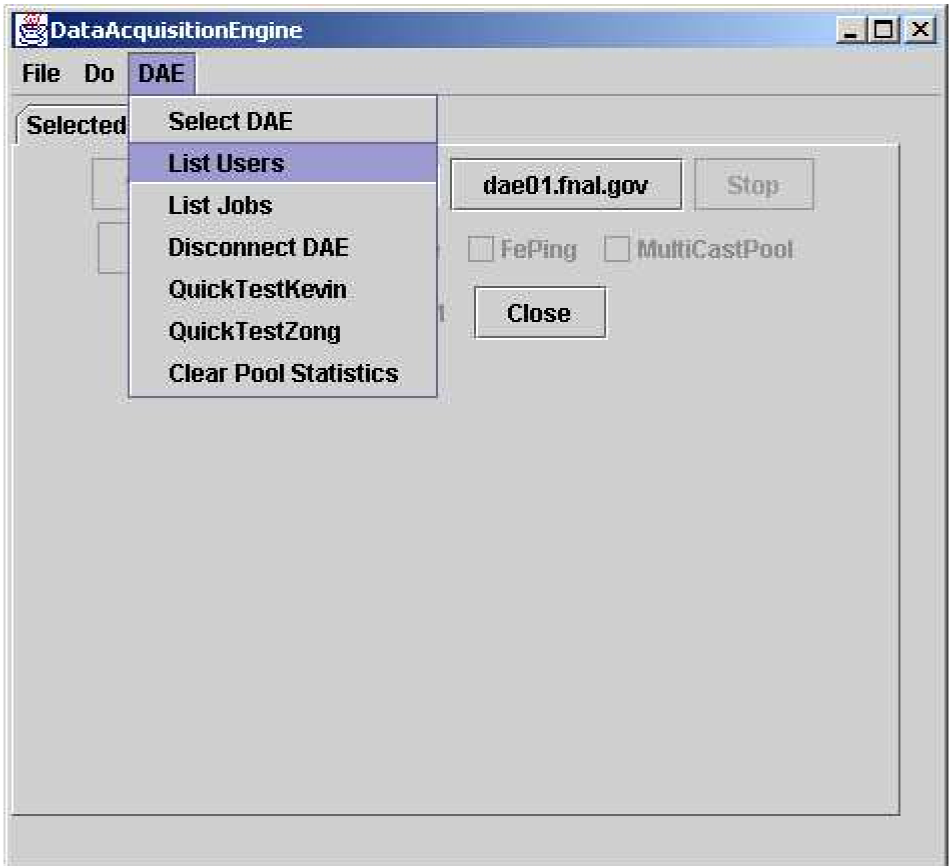
If the "Start" button is clicked, a stand-alone engine is started since the "Consolidate" button is not selected. If the "Consolidate", "FePing", and "MultiCastPool" buttons were selected before clicking "Start", then the engine will start up exactly as if started from OperationalRunEngine.

Consolidate means to access front-end resources through the DAE responsible for the front-end. FePing means that service pings for availability, Tevatron clock capability, and plot capability should be performed with the front-ends within this DAE 's responsibility. OneShot1 configures the OneShotDaqPool to acquire data one reading at a time, a useful performance measuring utility.

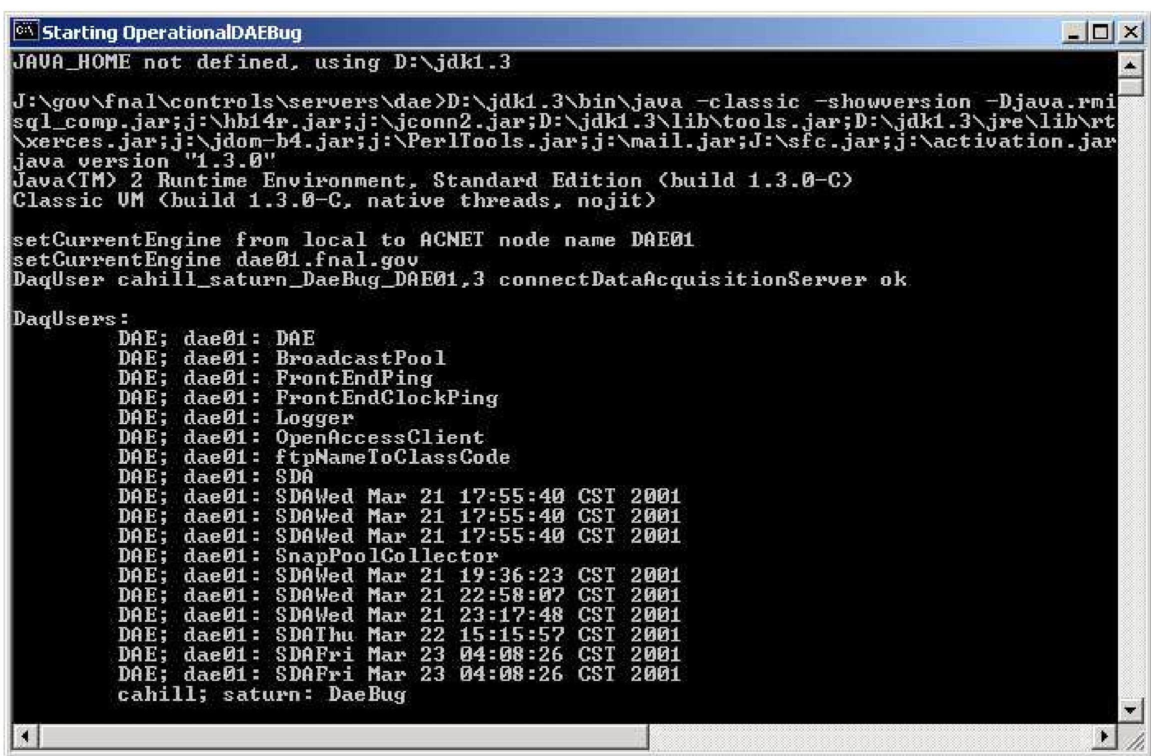
A DaeBug is useful even without starting an engine since it is still capable of running DaqJob (s). Any DaqJob started will try to connect to the engine listed on the button now labeled "local". Clicking on local will provide a Java ACNET node selector for choosing any Java ACNET node, and future job windows will be associated with the selected DAE.

A user may run several DaeBug applications connected to one or more server engines. Of course, only one instance may be running a local ACNET.

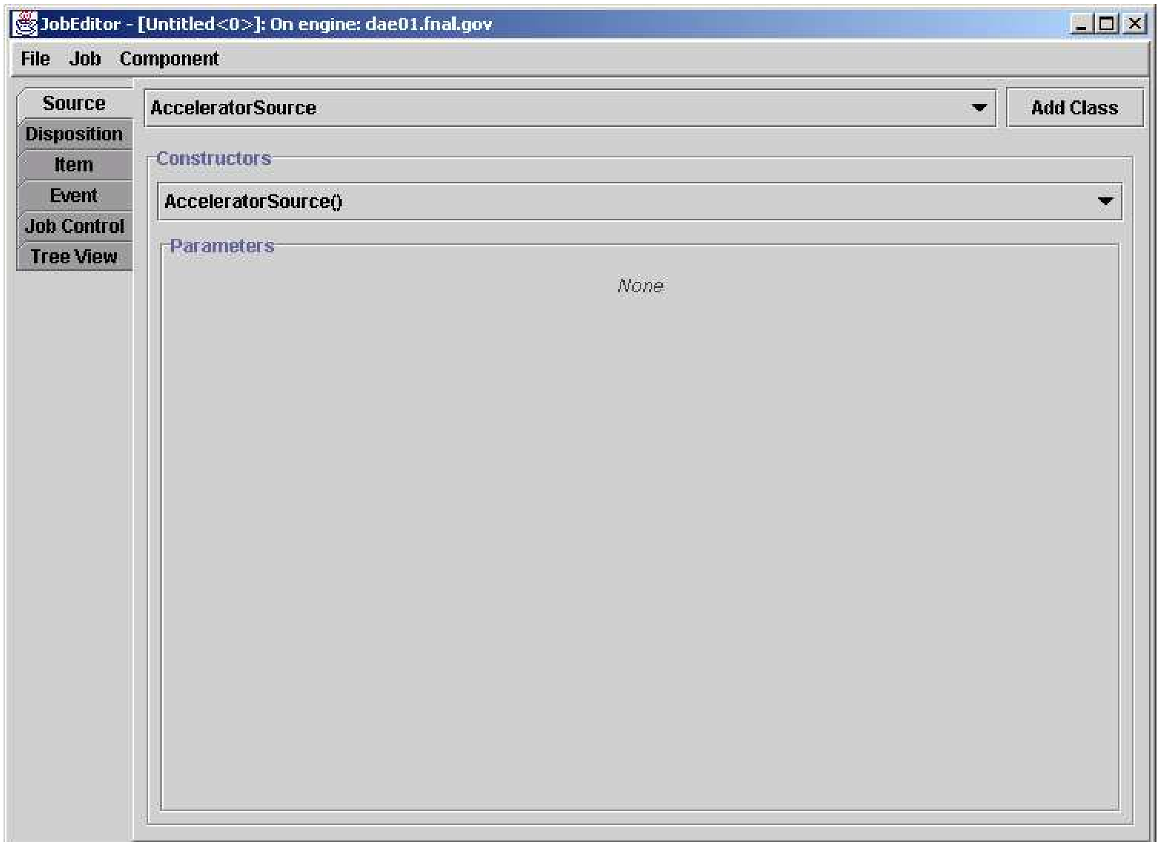
For example, choosing the node DAE01, and selecting the DAE menu item "list users"



will list the users connected to DAE01.



Selecting the “Do” menu item “NewCreateJob”
will pop a JobEditor window connected to dae01.fnal.gov.



The job editor uses the Java reflection API to find constructors and their arguments for source, disposition, item, and event. Some job components also provide picker functions for the job editor. Many types of DaqJob may be executed with this editor.

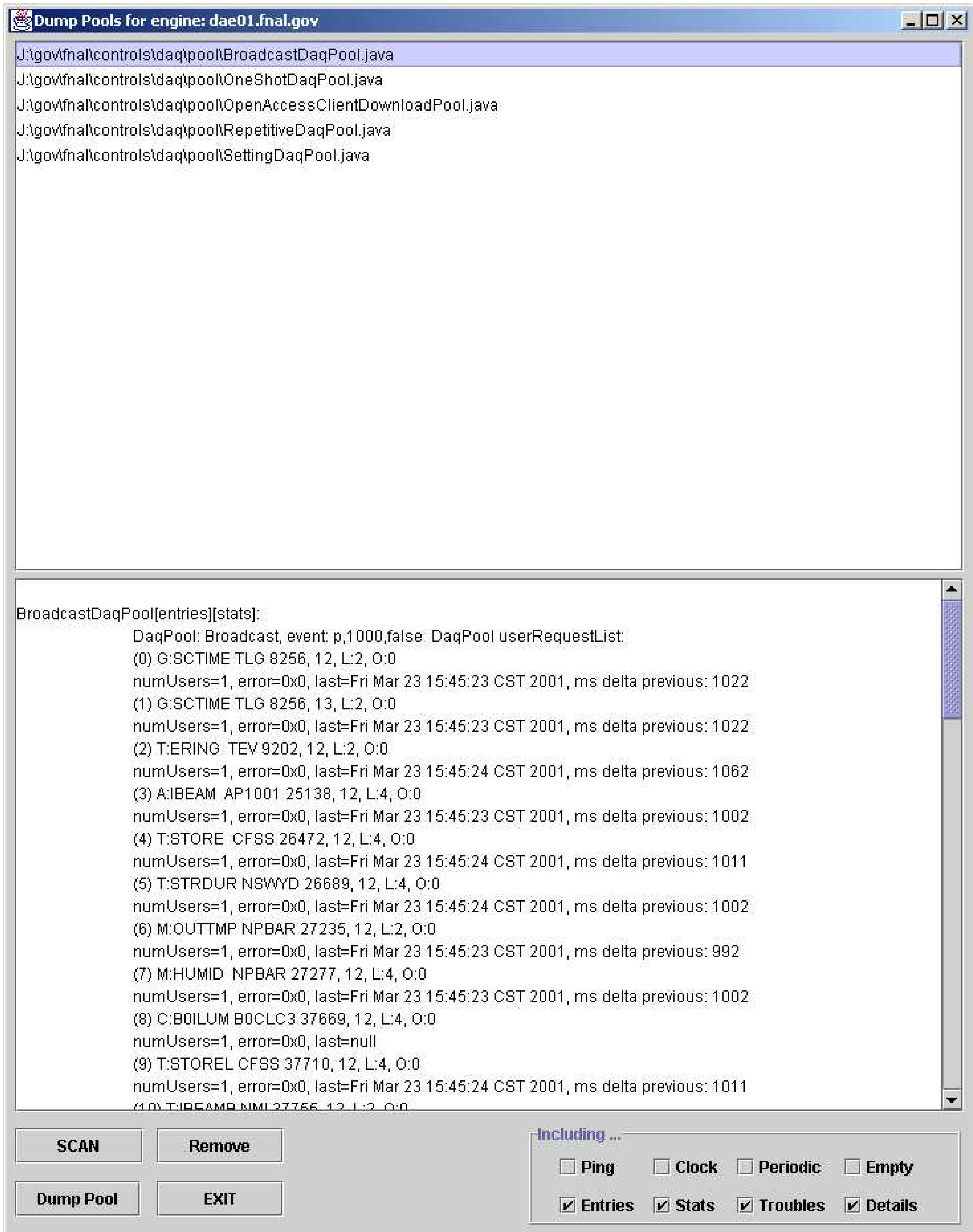
The Java reflection API is also used to aid debugging. It is often useful to compile a source module with a debugging flag turned on to report step-wise progression of data structures to terminal output as the module performs its function. It is also useful to report complaints or unexpected results to terminal output. However, a compiled debugging flag affects all users of that class for the lifetime of the class load, and terminal output complaints will eventually roll off the screen. It is more useful to be able to set the state of a debugging flag at runtime and to have the ability at runtime to report on statistics and complaints.

The reflection capabilities of the Java platform support the inquiry of supported methods in classes and the ability to invoke those methods dynamically.

Most of the engine’s classes support two static methods for setting and querying a debugging flag. The method isClassBugs returns a boolean state of the class’s debugging flag. The method setClassBugs takes a boolean argument to set the debugging flag. The engine's graphical user interface for the local engine supports a button labeled SetBugs that provides a file navigator to search for and add classes that support these methods to a window that supports setting and clearing the class's debugging flag. Any class that implements these two methods may use the SetBugs button to set or clear their debugging flag. The AcnetReadThread class for example will write to the terminal output a report of each incoming ACNET message when the class debugging flag's state is true. This functionality is available only on the local engine since setting the debugging flags on remote engines could harm their performance, and the output is not viewable by a remote engine user.

Similarly, most of the engine's classes support a static method names reportStatistics that takes no arguments and returns a string. The modules implementing reportStatistics maintain static variables with statistics about the class, including complaints and unexpected results. Most of these classes can return a long string with embedded carriage control that result in a report when this string is directed to a terminal output. The classes also typically suppress statistical counters with a value of zero. The engines support a menu item for ReportStatistics, and reflection and a file navigator are used to access to classes supporting reportStatistics. This functionality is available on the local engine as well as remotely to any engine in the control system since acquiring a statistics report is not assumed to cause any service interruption. An example of a statistics output:

Likewise, reflection is used to discover implementers of a static method dumpPools with 8 boolean arguments.



Most class authors would benefit by supporting these debugging features early in the development cycle. These features will aid the author and users in debugging and maintenance.

Engine Packages

gov.fnal.controls.acnet
Supports ACNET protocol, tests, and statistics. AcnetError defines the facility/error code status returns including translation to text, caching, and reports. AcnetMessage is useful for packing and unpacking binary data.

gov.fnal.controls.daq.accountability
Supports the accountability features of the control system. Who did what from where and when.

gov.fnal.controls.daq.acquire
Supports DaqJob, DaqUser and much of the RMI between the client and engine.

gov.fnal.controls.daq.bridge
Supports specific bridging functions from Java to the VAX/VMS control system. For example, informing the sequencer on front-end state transitions.

gov.fnal.controls.daq.bugs
Supports the debugging tools utilizing the Java reflection API for classBugs, reportStatistics, and dumpPools.

gov.fnal.controls.daq.callback
Defines the interfaces for many data acquisition callbacks associated with the components of DaqJob.

gov.fnal.controls.daq.consolidate
Engine startup, consolidation assignment, front-end heartbeat, and persistent jobs; what makes an engine an engine.

gov.fnal.controls.daq.context
Optional data return collection context, i.e. from which SDA or save file when source spans multiple sources.

gov.fnal.controls.daq.datasource
The source (from) and disposition (to) components of DaqJob.

gov.fnal.controls.daq.db
Support for database device type, index, and name.

gov.fnal.controls.daq.errors
Support for the logging and display of all errors across the control system.

gov.fnal.controls.daq.items
The item (what) component of DaqJob.

gov.fnal.controls.daq.logger

Support for data logger device and rate logging specification.

gov.fnal.controls.daq.mcast
Support for data acquisition multicast of popular channels.

gov.fnal.controls.daq.oac
Support for building Open Access Front-end Clients.

gov.fnal.controls.daq.pool
Data acquisition data pools, support of RETDAT protocol.

gov.fnal.controls.daq.scaling
Support for transformation of raw data to engineering units.

gov.fnal.controls.daq.schedulers
Engine realization of the DaqJob.

gov.fnal.controls.daq.security
Security authorization, control, and display.

gov.fnal.controls.daq.snap
Fast-time plot and snapshot plot protocol support.

gov.fnal.controls.daq.tools
Tools; job progress, device setting, …

gov.fnal.controls.daq.tree
Support of managed tree display of directory like information.

gov.fnal.controls.daq.util
Utilities, engine shutdown, pooled database connections, ACNET error definitions.

Contributing to Engine Packages

Reusable Job Components

The important components of a data acquisition job are source, disposition, item, and event. The thin client architecture encourages the development of new job components and extensions of existing components.

An ideal application consists of a single job. That job may be composed of other jobs and any job may start additional jobs. A job component of the application may be very application specific. For example, a parameter page application is defined by a job with a parameter page disposition. The traditional concept of a parameter page is encompassed in a job with a data source from the accelerator, a data item describing a page, category, and sub-page, and a default data event item. The parameter page disposition by default provides a graphical user interface.

The parameter page disposition is reusable. Changing the data source to an accelerator save file or the data item to a front-end node, or the data event to a group of Tevatron clock events modifies the function, behavior, and utility of the parameter page. It also reduces the need for custom displays, diagnostics, and duplicative applications. Overall, it reduces system complexity.

Likewise, a parameter page item is reusable. The item may be used as input to a database report, an alarms bypass page, or a Save/Restore save.

The plot display also exists as a disposition. Varying job components extends the plot display functionality to encompass fast time plots, slow time plots, snapshot plots, data logger plots, and others.

Other job components may be extended to provide a reusable job element exploited in new jobs to provide functionality not present in the existing control system. The data source CompareSources uses two sources of data for reporting to the disposition the data item elements that are different in several configurable formats. This data source can be employed to compare save files and report differences to a parameter page. It can be employed to compare the database download records to front-end settings. It is a reusable component that may be applied to a variety of jobs.

The Tevatron clock display, today's application page D33 Clockscope, might exist in Java as a disposition. Its constructor might include the display of clock events graphically. It might be imagined that this disposition has functionality even when its display is suppressed. The disposition could provide an interface for the specification of returned information about the current cycle, previous cycle, numbers and times of observed events, etc. Changing the data source to a data logger of Tevatron clock events or changing the data item to a subset of all Tevatron clock events are ways of extending the functionality of this reusable job component.

Complex accelerator concepts will often be implemented as a disposition. The disposition will often provide a graphical user interface. A typical data item suitable for a complex accelerator concept will often be the default data item, the job's means of relieving the user of the knowledge of how to 'feed' the disposition. In this case, some other job element is expected to provide data acquisition specification for the job. Likewise, the job's data event will often be the default data event.

Whenever possible, the job elements functions are performed within the data acquisition engine instead of within the client. When the data source is an accelerator save file, the save file is opened, read, and the data is scaled within an engine. When an item is a complex computational device item, the collection and computations are performed within the engine. Job elements within the engine have access to binary data, typically reduce the RMI overhead, and contribute to thin clients. The disposition in the engine is coupled with the disposition in the client with RMI calls often specific to the function of the disposition. For a beam position monitor disposition, the engine shares results to its client counterpart through RMI. These RMI functions are likely beam position specific, implemented only in this disposition, and providing the client access to beam position objects, results, and control. This obviates the need to encapsulate all complex devices within a single data acquisition model.

For job elements to maximize their reusability, they should insofar as possible be completely described by one of their constructors. That is, they should not have to be manipulated after construction by a knowledgeable class, as that reduces their reusability. Job element extenders should always keep in mind that their functionality is not targeted for a single application, and their ultimate reuse is expressed when exploited by a sequencer application to operate the complex.

A significant tool aiding the development and testing of job components is the job editor. The job editor's role is to create and run nearly any job within the control system. The job editor provides a visual interface able to construct any job component. The reflection capability of the Java language, coupled with guidelines to job component authors provides the capability to instantiate job components and to build and test jobs. The job editor also provides support for persistent job components, i.e. saving job components to a database including the ability to save jobs.

The ultimate goal is to imagine a complex, multi-functional application that can be described by a single job composed of jobs, and each of these jobs and their job components were created by the job editor.

Extending Job Components

The data acquisition architecture of the Data Acquisition Engine (DAE) encourages programmers to extend the components of the DaqJob, both for functionality and for callback.

Source, disposition, item, and event are the key components of a data acquisition job. They are literally subclasses of DataSource, DataSource, DataItem, and DataEvent respectively. They are passed by value to an engine using Remote Method Invocation (RMI). Each base class is abstract and each subclass must implement some functionality to satisfy the data collection requirements of the engine.

A programmer extending one of these abstract job classes to provide additional data acquisition functionality has several requirements. The programmer must implement the abstract methods of the subclass extended. The new class must be serializable as it will be passed by value to an engine when starting a job. The attributes of an object of this class necessary to fulfill the method invocations on the engine must be serializable, and the attributes not necessary to the engine should be marked transient to reduce serialization overhead and maximize effective network bandwidth.

A more common example of extending these abstract job classes is for the purpose of establishing callbacks. Interfaces define the data acquisition callbacks, and simply extending a subclass of DataSource or DataItem establishes the object as having implemented data acquisition callbacks.

The interface GenericCallback requires the interface implementer to implement methods to receive data acquisition callbacks for the generic properties of a device. A job's disposition, item, and item element may implement GenericCallback to receive callbacks in the disposition, item, or item element respectively. When extending MonitorChangeDisposition, AcceleratorDevicesItem, or AcceleratorDevice for the purpose of implementing GenericCallback, the serialized job argument to the engine needs no additional information other than identifying that this class implements GenericCallback.

When the engine receives a job argument, the Java ClassLoader loads the class from CLASSPATH typically or from the java.rmi.server.codebase URL specified at engine startup. Class discovery and loading are cached within the Java virtual machine and will not be reloaded until every reference to a loaded class has expired or a new class loader is installed.

When a loaded RMI stub class is changed by adding or deleting attributes (since the job arguments are passed by value), the engine's Java virtual machine will throw a java.rmi.MarshallException. This means a previous version of this modified class has already been loaded into the engine, and this engine cannot process this job argument. At this point, the engine could be restarted or the engine could delete and recreate a ClassLoader. Creating a new ClassLoader would result in users of the previous version (including the engine itself) receiving MarshallExceptions when they tried to use the older class. The analogy to older control systems is passing a C structure by value, having programs running using that C structure, modifying the C structure and trying to run old and new users of the structure at the same time in a single server. Java is an improvement in that it can handle an object that it has never seen, that was created after the virtual machine was run, but without extraordinary steps will not handle multiple definitions of an object.

The organization of engine clusters exists for this reason. The operational cluster will reference an unchanging class library. The test cluster will reference an unchanging beta class library. Most data acquisition users extending job arguments do not need to include their class attributes in the object passed by value to the engine. Care needs to be taken to mark those attributes as transient, both to reduce unnecessary serialization and network traffic as well as eventual MarshallException problems when adding and deleting attributes while developing and testing with some engine. The promotion of thin-client extensions to job elements includes the support of a stand-alone engine for the developer. Extensions to job elements may be created, modified and tested without restarting the local engine. Only, when a particular class' RMI signature of the serialized object has changed, and a previous version was loaded, will the local engine have to be restarted.

Of course, especially early in development, sometimes a base class is modified due to an oversight or needed extension. This will manifest itself in any engine that has loaded that base class and is seeing through RMI the changed class. Restarting that engine is required to fix the MarshallException.

There are no plans at this time to write a private ClassLoader or reload a ClassLoader. The disadvantages appear to outweigh the advantages.

Complex Extension of DaqDataItem

The example is based upon the implementation of CompDeviceItem, a computational device item. The item knows its input devices. The source, disposition, and event are unimportant since this subclass of DaqDataItem must perform for a variety of job component specifications but assume the source is AcceleratorSource, the disposition is a subclass of MonitorChangeDisposition, and the event is repetitive.

This item implements SchedulerState so to be informed of schedulerBegin, receiving a reference to the DataScheduler which allows it to learn of and remember the disposition through getDisposition. The disposition must be informed of completion of data acquisition intercepted by this item.

The key method the DaqDataItem implements is whatDaqs which returns a Vector of WhatDaq for data acquisition. This method is passed and remembers the implementer of ReceiveData (typically an interface the disposition has implemented) to push into the WhatDaq and a DataSchedulerInterface identifying the RMI connection with the client job to push into each WhatDaq. This method is also passed an initial id to populate and increment an id tag in the WhatDaq. The disposition expects to see these ids in its ReceiveData interface, but since this item will intercept ReceiveData, these ids are used to report completion to the disposition.

This item implements ReceiveData and pushes 'this' into each WhatDaq it requires for input. Likewise, it pushes and increments the id and pushes the scheduler interface. The Vector of WhatDaq returned includes all the necessary collection elements for the item but will not include a WhatDaq of the resultant device.

On receipt of raw data in ReceiveData, this item informs the disposition of element completion by calling the disposition's completionCheck passing the WhatDaq id. Next, the generics method in WhatDaq scales the data. The item implements GenericDAE which is the engine equivalent of GenericCallback. The item receives readings, computes a result, unscales the result to raw data, and invokes the remembered ReceiveData with information about the resultant device. The resultant device, a WhatDaq created on the fly must populate its genericIndex from this item's genericIndex inherited from DaqDataItem. Ideally, the disposition should not be informed of completion twice for one item, so one of the ReceiveData invocations should not report completionCheck, and that id should be used when invoking the remembered ReceiveData.

Job Component Support for the Job Editor

The components of a data acquisition job are source, disposition, item, and event. Each of the job components strives to maximize reusability. Many of the job components subclass other components to add functionality and flexibility. More importantly to this note is the reusable nature of the job component in a job. A front-end node item for example may serve as a collection specification for a save file, parameter page, or report disposition. It may serve as a destination specification for a download or restore disposition. Likewise, a single plot disposition will provide slow, fast, snapshot, and data logging plots.

The job editor provides the capability to specify job component creation through selection and editing of the argument list for each of the component's constructors. A front-end node item has two constructors, one based upon ACNET trunk and node and one based upon ACNET node name. The job editor supports the selection of the constructor and the specification of the constructor's arguments. The component or the entire job specification may be saved to an XML file that may be used to instantiate the job components and start a data acquisition job.

To illustrate the difference between the VMS control system and the job architecture, consider an alarm's bypass control application. In the existing control system, the application maintains lists of devices to query and control. The lists tend to be private to the application and limit the application's functionality. In the job architecture, the alarm bypass control application is represented by a job disposition. The collection of devices may be specified by a data item referring to similar database lists, but might also be specified by a front-end node item, a save subsystem list id item, or a set of parameter page data items. Also, changing the job's data source to a save file, for example, extends the disposition's functionality. Lastly, the database lists of devices used for frequent alarm bypass scans are available as a data item to the parameter page or other dispositions.

Many functional elements of the existing control system may be replicated by defining a job using the job editor, making the job definition persistent as an XML file, and providing a command file or link to start the job. Additional functionality is available through the modification or creation and running a job in the job editor.

The job editor serves several communities.

The job editor serves the programmer by providing access to every job component constructor and its arguments. A job component may be instantiated with nonsense argument values, and a job may be tested with illogically grouped job components.

The job editor serves the application builder as generally the application's concept is represented by a job disposition, and the default application behavior is determined through the specification of other job components.

The job editor serves the application users by providing the ability to create and execute jobs from components of their choosing.

Job component authors must adhere to some principles to maximize the reusability of their component and the job editor.

A job component constructor should exist that fully specifies the component, as the job editor will not provide access to methods that may change the behavior of the component.

A job component constructor argument should provide for temporal consistency. If a data logger node is required as an argument, specify that argument as a DataLoggerSource rather than as an ACNET trunk and node, for example as the data logger and its data may move to another node. Likewise, a constructor should provide for a relative concept of time, not simply absolute Date objects.

The job component should provide a static constructorArgumentsPicker function when the errors and inconsistencies may be reduced by such a function. An AcceleratorDevice constructor includes device name, property, length and offset. The job editor user has the opportunity to specify many AcceleratorDevice objects that do not represent the constraints of the device database. For device M:OUTTMP, only a length of 0 or 4, a property of 1 or 12, and an offset of 0 will result in successful data acquisition. The constructorArgumentsPicker for AcceleratorDevice would provide the job editor user a constrained returned argument list.

Internals

The following sections contain a more detailed explanation on a topic.

Engine Startup

The engine startup package is gov.fnal.controls.daq.consolidate. The Monitor object when instantiated defines engine behavior. An operational clustered engine has the boolean arguments consolidatedEngines, pingFrontEnds, and multicastPools set to true. A stand-alone engine instantiates Monitor with each of these arguments set to false.

Consolidated engines indicate that the engines in the cluster are assigned front-end responsibility and other engines seeking data acquisition from a front-end obtain the data through request to the responsible engine. The responsible engine pings the front-end for up status, clock event read capability, and fast time plot protocol support.

The front-end status ping requests a reading from the guaranteed readable device defined in the ACNET node tables (page D98) at 0.1 Hz and restarted each minute. When a front-end fails its ping, the front-end node is multicast as down using a state transition device. The consolidating engine backs off all ACNET message retries except the ping request and informs outstanding and future requests that the node is down. On ping success, other ACNET requests are resent. All engines maintain the status of all front-ends as a service to interested clients and as preparation to server as a backup engine.

The front-ends are pinged for clock event read and fast time plot and snapshot plot protocol support. An engine begins with these services defined by the ACNET node table definitions, but changes their status dynamically. When a front-end does not support these services, the engine supports them with engine software.

Engines ping each other and declare engines down using a state device. Engines ping engines ahead of them in the ACNET node tables until finding a responding engine. Intervening engines' front-end responsibilities are assumed by the pinging engine. Since engine status is multicast, other engines quickly discover the responsible, backup engine. When the non-responding engine is back, its state transition announcement causes requests served by the backup engine to restart.

Engines have input and output files. Their path begins with the engines.files path at the same level as the gov tree. Directories and subdirectories exist by function. Two sets of directories exist: engines.files for the operational cluster and engines.files.test for the development cluster. When a consolidated engine starts up, it reads the file engines.files.test.startup.TestEngines to determine if its node belongs in the development cluster.

Engine startup is somewhat complicated. Startup speed is important, and many initializations must complete, and some are dependent upon each other. Several threads are started with thread blocking at appropriate points. Startup issues include reading the ACNET node tables, initializing the ACNET service, cluster determination, Open Access host and client initialization, finding ping devices including strategies for incorrect or missing entries, transmitting ACNET killer signals to all front-ends to cleanup any traffic from a previous run, learning engine consolidation assignments, preparing to listen to multicast clock events, state transitions, and the multicast data pool, setting up to honor RMI requests from clients, and starting persistent clients such as data loggers, Open Access clients and models.

DaqJob Detailed Flow

The following describes in greater detail the flow of the DaqJob obtaining a single reading of outdoor temperature.

Beginning at job.start(), execution is in the context of the client, and the item is first job component under scrutiny. The item is directed to establish callbacks.

Remote method callbacks to return the reading must be established before sending the job components to the server. When acquiring traditional accelerator data, the disposition is likely to extend MonitorChangeDisposition and implement GenericCallback. In this case DaqEngineTerminalOutputDisposition does both. In the simplest case, a disposition extending MonitorChangeDisposition and overriding the reading method is sufficient to obtain the outdoor temperature reading.

The application programmer is not expected to directly support remote objects. The item calls GenericEstablishCallback to map the user's callback to a remote callback the item establishes, publishes the remote object, and passes the remote object in the disposition. The user's disposition is required to implement GenericCallback for this type of job, and the programmer may receive the reading by overriding the reading method. When the job is composed of several items, or the item contains several devices, it may not be convenient to receive all the callbacks within the disposition object. When an AcceleratorDevice implements GenericCallback it overrides an item that implements GenericCallback that overrides a disposition that implements GenericCallback. In fact, a job can be composed such that devices, items, and the disposition each receive reading callbacks.

Before the DaqUser module sends the job components to the server, it ensures the connection to the server exists. Connections can be reestablished at this point, and jobs can be refused because of inadequate user privilege. Each of the job components on the client side is notified that the job is beginning, and the job components are sent to the server.

The DataAcquisitionSupport module in the package acquire on the server receives the job components sent via RMI by the client. The server verifies privilege and instantiates a scheduler suitable for the set of job components or throws an exception. For this job, an AcceleratorPoolScheduler is created and a DataSchedulerInterface is returned to the client for canceling the job or receiving job completion notification. The scheduler informs the job components on the server side that the scheduler is beginning the job. Every AcceleratorPoolsScheduler has an item that extends DaqDataItem that will return a vector of WhatDaq that define the devices to collect. The number of devices to collect is passed to the disposition where the base class DataSource supports counting down disposition elements to determine job completion. The scheduler inserts each data acquisition request into the appropriate OneShotDaqPool, processes the pools and waits for the disposition to tell the scheduler that the job is complete. The pools build RETDAT frames to send to send via ACNET to the engine consolidator for the front-end or directly to the front-end. The front-end eventually sends ACNET packets back to the consolidator who completes a consolidation job by sending an ACNET packet back to the server who scales the data and processes the RMI callback to the client. The RMI callback code is in the GenericEstablishCallback calls the mapped user callback. The scheduler informs the server side job components of scheduler completion, the client side job of job completion, and the client side informs the job elements of job completion.

A single application may have multiple connections to an engine. On an initial DaqUser connection from a user node to an engine, the engine looks up, caches, and downloads to the client a Privilege object. During the life of the DaqUser connection, the engine pings the client, and if the connection is lost, stops all jobs associated with that user. The client also pings the engine, and if the connection is lost and regained optionally restarts jobs dependent upon the DaqJobControl object of each job. The DaqJobControl also provides interfaces for informing the client of job trouble, statistics, and completion.

DataSource and disposition

AcceleratorSource and AcceleratorDisposition support reading and setting devices through their front-ends.

BroadcastPoolDisposition is the source of the multicast pool frames. The bandwidth necessary to share a popular channel has moved from the front-end to the consolidating engine. The database supports the definition of several broadcast pool ids for each device. Currently, most of the channel 13 devices have a 1 Hz broadcast pool id defined. Consolidating engines responsible for the front-end of a broadcast pool device start a DaqJob to collect the device. The BroadcastPoolDisposition passes the readings on to a multicast frames for multicasting to all engine nodes.

CallbackDisposition is an abstract class supporting the RMI calls necessary for the GenericCallback implementation. MonitorChangeDisposition is an example of a subclass of CallbackDisposition.

CompareSources supports the return of one, the other, both, or the difference of two data sources supporting the comparison of a save file to real time, two save files, real time and the database download settings, and many other combinations.

CorrelationDisposition supports the correlation of reading or plot data from multiple front-ends.

DaqEngineTerminalOutputDisposition utilizes System.out for reporting job output. Since this disposition implements most callbacks, it is a useful disposition for debugging and code review.

DataLoggerClientLoggingDisposition supports Open Access Client data logging by providing simple methods to forward client data repository changes to an associated data logger.

DataLoggerSource supports jobs collecting data from a data logger.

DataLoggerDisposition is a data logger.

FermiDataDisposition returns data to the user in the form of an object.

JobDataSource uses a DaqJob as a data source supporting jobs containing jobs.

KnobUserSettingSource supports parameter page knobs.

ModelSource and ModelDisposition are used when directing data acquisition from and to an operational model.

MonitorChangeDisposition is a popular disposition to extend for users implementing GenericCallback. Supports returns only on changed data or error as well as return all callbacks.

ObjectCallbackDisposition is used to return an object.

ParameterPageDisposition is the parameter page application.

PlotDisposition is a convenience disposition for users requesting plot callbacks.

SavedDataSource and SavedDataDisposition support save files.

SDAUserViewDisposition extends ParameterPageDisposition with a complex but partial implementation of TableModel.

SlowPlotDisposition and SnapPlotDisposition are plotters.

TestSource and TestDisposition support reading from and safe setting to front-ends and Open Access clients under test through redirection.

TreeDisposition displays its result as a Tree.

UserSettingSource is the repository of user setting data.

DataItem

AcceleratorObject, AcceleratorDevice, AcceleratorDevicesItem support name or device index, property, length, and offset.

AcnetNodeStatusItem identifies the nodes to monitor for ReportCallback status returns.

BeamPositionMonitorScalingItem works with a VAX/VMS process that scales and returns beam position monitor readings.

BigPingItem is the collection of all front-end ping devices if this engine was consolidating on behalf of all front-ends, i.e. all operational and test front-ends and Open Access clients.

ColliderShotItem, E835StoreItem, and PbarTransferShotItem are the input definitions for their respective Sequenced Data Acquisition collections.

CompDeviceItem supports dynamic devices composed of an expression containing accelerator devices.

CompoundDaqItem supports a collection of DataItems. For example, several save subsystem list ids and several AcceleratorDevicesItems containing display list devices.

CompoundDataloggerItem supports the retrieval of data logger data.

CompoundFTPItem supports the retrieval of fast time plot data.

CompoundSnapItem supports the retrieval of snapshot plot data.

CompSetDeviceItem supports settings to dynamic devices composed of an expression of accelerator devices.

DaqAllItem is the all device for some context. In the context of Save/Restore save it is the collection of devices for a big save. For a Save/Restore file, it is all the devices in the file.

DaqEngineDebugTestItem was the early job test module for the engine. Several static methods demonstrate many varieties of DaqJobs.

DaqJobsItem supports a collection of jobs for a job containing jobs.

DataLoggerListItem the collection of devices by list id within a data logger.

DataLoggerRetrievalItem supports the retrieval of data logger data.

DefaultDaqItem is the default data acquisition item. It is context sensitive. For a ParameterPageDisposition job, the default item would be the last ParameterPageItem referenced by that user.

DeviceIndexArrayItem supports an array of device indices.

DeviceNameArrayItem supports an array of device names.

ErrorListItem supports the retrieval of error logging data.

EventDataItem identifies clock and state transition events for a job monitoring events.

FTPDataItem is an abstract class supporting fast time plot requests.

FrontEndNodeItem identifies all the devices addressed by a front-end.

HelloWorld item is a test item that implements OutputTrigger.

ObjectItem is an abstract class that other items identifying objects extend.

OpenAccessClientItem is the FrontEndNodeItem equivalent for Open Access clients.

ParameterPageItem supports the devices on a particular parameter page and sub-page.

PlotDataItem is an abstract class supporting plot callbacks.

ReportItem is the abstract class supporting report requests.

SDADataItem is the abstract class supporting sequenced data acquisition.

SDAReportItem defines a request for SDA save file reports.

SDAUserViewItem contains a collection of devices on a sub-page of the SDAUserViewDisposition application.

SaveFileReportItem supports the retrieval of directory information about save files.

SaveListIdItem refers to the collection of devices with a specified database save subsystem list id.

SavedDataAreaTreeItem supports the retrieval of SDA and Save file data.

SnapDataItem supports a snapshot plot request.

TreeItem is an abstract class defining a request for a mutable tree.

WhatDaqVectorItem supports the retrieval of a Vector of WhatDaq objects..

DataEvent

AbsoluteTimeEvent defines an absolute time.

ClockEvent defines a Tevatron clock event.

DataLoggerClientLoggingDataEvent is probably unused.

DefaultDataEvent defines the default event for the context of the job. For example, a parameter page job uses this event to specify that data collection frequencies should be the defaults specified in the database.

DeltaTimeEvent defines a time between events. It can define a periodic rate as well as a T1 – T2 event.

EmptyDataEvent specifies a null event. Event information should be found in another job element. SDA for example has event information embedded within the item.

KnobSettingEvent defines an event in support of settings where the resources in the client and server are maintained between settings.

LoggerRetrievalEvent supports data logger retrieval options for skipping points, specifying a minimum time between points, and date alignment.

MCClockEvent is a multicast clock event supporting clock decoding.

MonitorChangeEvent is an unimplemented description of how monitor change should be implemented.

MultipleImmediateEvent defines an event that occurs immediately and is considered repetitive. Used for knobbed settings.

OnceImmediateEvent defines an event that occurs once and immediately.

SavedDataEvent defines a collection of saved data file and collection indices.

StateEvent is a software state transition event.

Data Pools

WhatDaq

The WhatDaq is a key data structure supporting data acquisition. It holds the device name and index, property, length, offset, frequency, broadcast pool ids, front-end, SSDN, scaling object, RMI callback, disposition completion id, and a myriad of other job and data acquisition parameters.

SharedWhatDaq

A SharedWhatDaq contains a user list allowing one SharedWhatDaq to hold references to many WhatDaq objects. Data acquisition consolidation by device within a pool is the role of the SharedWhatDaq. Device consolidation is by unique by length, offset, and frequency.

Adding Pool Requests

The AcceleratorPoolScheduler locates the appropriate pool and inserts an item's WhatDaq requests into the pool. The pool adds a WhatDaq to the user list in a SharedWhatDaq, creates the SharedWhatDaq when necessary.

Processing a Pool

Pools support user request lists containing SharedWhatDaq objects with user WhatDaq objects randomly added and deleted. Meanwhile the front-end and pool are working with an active request list containing SharedWhatDaq objects from the user request list when the pool was processed causing a new set of network data structures to be shared by the front-end and pool.

Delivery of Data

The network data structures sent to the front-end return ACNET replies containing data to the pool. The pool traverses the active request list of SharedWhatDaq objects delivering data each entry in the SharedWhatDaq user request list to the address specified in the user WhatDaq.

Front-end Consolidation

When the engine is a member of a consolidating cluster, the ACNET frames are sent to the RETDAT task in the front-end only when the engine is the consolidating node for that front-end. Otherwise, the ACNET request with an additional header is send to the task POOLER on the consolidating engine.

BroadcastDaqPool

The device database may contain broadcast pool ids for each device where each pool id corresponds to a broadcast data pool updated at a specified frequency. Very few of the devices in the database have specified broadcast pool ids. The broadcast pool exists when the system load to multicast device values is considered to be less than transmitting ACNET replies to each engine requesting a specific device from a consolidator engine.

As each engine starts up or assumes responsibility for another engine, a job is started to collect data for devices whose front-end consolidator is the local engine at the specified frequency and multicast the data returns to all engines.

Engines receiving the multicast frames deliver data to the broadcast pool in a similar manner as the ACNET delivery from a front-end. The broadcast pool contains a user request list and active request list and operates as a repetitive pool.

The AcceleratorPoolScheduler checks each WhatDaq to determine if collection can be satisfied through insertion to the broadcast pool. Currently, the return frequencies must match exactly, i.e. a 0.5 Hz request is not served by a 1.0 Hz broadcast pool.

OneShotDaqPool

The one shot pool manages potentially large FIFO priority ordered queues. Users are generally not expected to pace or manage resource allocation for large requests. A one shot read of the Tevatron front-end is a reasonable request even if it results in 25,000 readings. The one shot pool manages first in, first out queues, pacing the length of the active request list to accommodate the front-end and sending another request only when the previous has completed.

Any user request added to the tail of a previously large request will wait some time for the FIFO queue to shrink to the point the user request is fulfilled. Consequently, there are three priority ordered queues where the lowest priority queue is used for large data items, such as FrontEndNodeItem and SaveListIdItem. The high priority queue is used for soft event collection and SDA data items. Otherwise, the normal priority queue is used. On each return of one-shot data, the next request is from the highest priority queue, and only one queue is serviced at a time.

The one shot pool also differs from repetitive pools in that it retries collections on some errors. Errors are categorized as retry-able, resource limiting, or final. Retry-able errors are retried. Resource-limiting errors initiate retries with a smaller number of requests if possible.

PoolPacketAssembly

Users are not expected to confine their requests to size limitations of the ACNET protocol. Pool packet assembly allows users to request up to 32K bytes of data. The data is collected using linear addressing across multiple ACNET frames, assembled, and delivered to the user.

RepetitiveDaqPool

The repetitive data pool serves repetitive data acquisition requests. Simple repetitive request rates such as 1 Hz are satisfied with a multiple reply ACNET request. Requests for return on clock event are satisfied in the same manner if the front-end supports return on clock event.

The repetitive pool operates in multi-shot mode when collecting on clock event to a front-end that does not support the Tevatron clock and for longer periodic rates. Multi-shot mode involves local timers to fire and initiate a one shot collection.

Only engines serving as the consolidator of a front-end employ multi-shot pools. When 25 engines servicing the same device with a collection rate of 10 minutes, a single one shot collection from the front-end satisfies all engines since they treat repetitive requests for front-ends not owned by themselves as simple repetitive. Engines shifting front-end responsibility as other engines come up or go down must shift some pool behavior from multi-shot to repetitive and vice versa.

The repetitive pools attempt to maintain front-end data returns by timing out replies, restarting requests, and toggling the front-end active states. As a front-end enters the down state, all pool requests except the ping pool (0.1 Hz) return an error and will not restart until the front-end declares itself or is declared up.

SettingDaqPool

The setting pool is an extension of the one shot pool. Priority is undefined for the setting pool. Any new setting with a setting on the FIFO queue completes the earlier setting with a positive error code. This prevents undisciplined knobbers from filling the FIFO queue to a slow front-end.

Schedulers

All schedulers extend DataScheduler, an abstract class that provides a good deal of functionality to all schedulers. Schedulers are runnable. The engine starts and runs a thread of execution on behalf of a job. The job ends when the scheduler thread ends. Schedulers, in general, do not do much work, but coordinate the job components who do the work.

The DataScheduler constructor establishes signals for job setup and completion coordination, publishes itself as a remote object to be returned to the client for future job cancellation for example, and notifies each of the job elements of scheduler begin.

The run method is implemented in each subclass of DataScheduler and is the existence justification of the scheduler. When the run method completes its scheduler specific calls, a call to finish in DataScheduler coordinates blocking the thread until the disposition is satisfied the job is complete. DataScheduler informs the subclass of scheduler completion, informs each of the job components of scheduler completion, informs the client's job of scheduler completion, removes this job from the engine's job list, and exits the thread.

AcceleratorPoolScheduler

The AcceleratorPoolScheduler works with data pools that work with the front-ends.

Items are a subclass of DaqDataItem guaranteeing they all return a vector of WhatDaq and initialize each WhatDaq with a unique, incrementing id for disposition completion, stuff the reference to the interface handling raw data, and a reference to the scheduler. The number of expected raw data returns is shared with the disposition, and the disposition will inform the scheduler of completion on receipt of each unique, incrementing id. Each WhatDaq is inserted into the appropriate DaqPool. The list of affected pools is remembered and each pool is notified to process new requests. On a job cancellation, the remembered pools are asked to remove entries with this scheduler's reference.

ClientPoolScheduler

The ClientPoolScheduler class supports data acquisition for Open Access clients under test as well as Open Access models and models under test. The VMS OpenAccessClient architecture uses ACNET as the interface between the front-end server and the client so front-end servers and clients may run on different nodes.

The Java architecture will never use ACNET as a communication protocol between an engine and client. The initial implementation has the engine and client sharing an address space, and threading separates the client from the engine. This will be the best performing architecture, but will likely be followed with a client to engine RMI connnection, a CORBA to engine connection, and even perhaps a tcp/ip engine to client connection.

The initial Java Open Access scheme: An Open Access client and model are considered operational elements. Typical Open Access clients and models use the DaqSendThread interface, so the D92 OAS node, OAC node, and database entries in the case of clients should all point to the same operational engine. Models would have the same OAS and OAC node but have no database entries. The service id in the SSDN continues to address which client on that engine owns a particular device.

Open Access clients and models also need to be tested without interfering with operations. The DataSource for access to operational clients and models are AcceleratorSource and ModelSource respectively. Test clients and models are not supported in the operational DAE cluster. Starting a job with a DataSource of TestSource (models and clients under test) on an operational engine will yield an exception. Starting a job with a DataSource of TestSource on a non-operational engine will instantiate (if necessary) that client or model for testing. Several non-operational engines could have the same client or model under test. Applications will reference the same client or model data repository when their jobs run on the same engine.

Redirection under VMS whacks ACNET trunk/node and the SSDN. Models whack additional bytes of the SSDN to carry model specification information (Save/Restore file number, for example). The Java classes do not propose to connect to models and clients under test on VMS. The Java side does intend to support VMS application access to Java clients and models under test (with some modification of D128 and VAXDPM). The WhatDaq structure will be modified as little as possible. Since engine consolidation traffic will not occur for clients and models under test, the ACNET trunk/node and SSDN client class are not required to carry the addressing information.

Operational clients have the addressing information in place in the database. Operational models are subject to consolidation traffic and will have to force the RETDAT header to carry information or the WhatDaqs will have to be whacked similar to the incoming VMS traffic to an operational engine.

The data pools are keyed on trunk, node, and service id. For operational VMS clients and models as well as DAE operational clients and models, the trunk/node represents the OAS node and the service id is the SSDN class. Operational clients are included in consolidation traffic, so the ACNET RETDAT header coupled with the SSDN has to find its way to the correct data pools. Open access clients are straightforward in that the OAS trunk/node and the SSDN class will lead to the same pools on the consolidating and consolidator nodes. Operational models are more difficult. A model on the VMS side needs to have its SSDN whacked to the model's OAC class. A Java consolidating RETDAT message will carry the pseudo trunk and node in its frame to permit the reconstruction of the service id. A VMS RETDAT request would not have the header, but would have whacked the SSDN, so one can still find the correct data pool based upon service id.

Test clients and models will not be consolidated. The reasons include the usual scenario of testing new code in a stand-alone engine, of not wanting to stop and start a consolidator, not wanting to instantiate all test clients and models on the consolidating engine at startup, and the burden of extending the RETDAT header to contain test node and service configuration. When testing clients and models, data pools may exist for the operational clients and models, so the test client and model data pools must be keyed differently by trunk, node, and service id. The data pool key will use the pseudo trunk and node with a service id of zero.

ConsolidationScheduler

Data acquisition requests are consolidated across the control system to reduce front-end bandwidth. Consolidating engines not responsible for a front-end forward their ACNET RETDAT and SETDAT frames with an additional header to the engine responsible for that front-end.

The ConsolidationScheduler builds and submits a data acquisition job. It packs raw replies into an ACNET frame when complete sends a reply to the requesting engine.

DataLoggerScheduler

When a job's data source is DataLoggerSource, DataLoggerScheduler manages the retrieval of logged data by passing the PlotDataItem to the DataLoggerSource.

EventScheduler

When a job's data item is an EventDataItem, EventScheduler manages the monitoring of Tevatron clock and state transition events through the EventDataItem itself.

FTPScheduler

When a job's data item is an FTPDataItem and the job's data source is AcceleratorSource, FTPScheduler manages the collection of fast time plot data.

The run method acquires a ClassCode object if necessary for each plot request. Each FTPRequest is inserted into the appropriate FTPPool. The list of affected pools is remembered and each pool is notified to process new requests. On a job cancellation, the remembered pools are asked to remove entries with this scheduler's reference.

FileScheduler

When a job's data item is a TreeItem or the job's source is SavedDataSource or DownloadSource, FileScheduler manages the job.

TreeItem itself manages TreeItem requests.

Collection from a SavedDataSource and DownloadSource is accomplished by inserting user requests and processing them in a manner similar to data pools since these data sources implement the DaqPoolUserRequests interface.

JobOutputScheduler

When a job's data source is a JobDataSource, JobOutputScheduler manages the job by building and starting a job with job components extracted from the JobDataSource.

JobScheduler

When a job's data item is a DaqJobsItem, JobScheduler manages the job by building and starting jobs from the job components extracted from the array of JobSpec objects extracted from the DaqJobsItem.

ObjectScheduler

When a job's data item is an ObjectItem, ObjectScheduler manages the job by processing the request through the ObjectItem itself.

ReportScheduler

When a job's data item is a ReportItem, ReportScheduler manages the job starting report generation through the ReportItem itself.

SequencedScheduler

When a job's disposition is a SavedDataDisposition, SequencedScheduler manages the coordinating job. Operator saves, big saves, and sequenced data acquisition (SDA) use SDACollection and SDACollectSet objects to define and run data acquisition jobs in a sequenced order to accomplish the save.

Big save defines an SDACollection that will result in SDACollectSet job for each operational front-end. The SequencedScheduler coordinates how many front-end collection jobs are active at a time.

An SDA run defines many SDACollection objects with arming, collection, and ending events and other rules. The SequencedScheduler monitors the events and initiates SDACollectSet jobs throughout the run observing the SDACollection rules.

An operator save is a simple SDA with one SDACollection and SDACollectSet with simple rules resulting in a single OnceImmediateEvent collection.

SettingScheduler

When a job's data source is a UserSettingSource and the disposition is AcceleratorDisposition, SettingScheduler, an extension of FileScheduler manages the job. Settings' jobs can be thought of as two jobs. The first job is to extract the settings from the UserSettingSource and deliver them to the disposition. When the disposition is AcceleratorDisposition, the second job is to insert the delivered settings into the appropriate SettingDaqPool and process the pools.

SnapShotScheduler

When a job's data item is a SnapShotPlotItem and the job's data source is AcceleratorSource, SnapShotScheduler manages the collection of snapshot plot data.

The run method acquires a ClassCode object if necessary for each plot request. Each SnapRequest is inserted into the appropriate SnapShotPool. The list of affected pools is remembered and each pool is notified to process new requests. On a job cancellation, the remembered pools are asked to remove entries with this scheduler's reference.

Open Access Front-end Architecture

Introduction

Open Access Front-ends (OAF) are ACNET front-ends since they supports all of the ACNET data acquisition protocols. They differ from other front-end architectures in that they exist in the console and server layers of the control system whereas traditional front-end operate at the microprocessor layer. Consequently, all the facilities of the top two layers of the control system are available to the Open Access Front-end and its clients. Additionally, the Java clients run within a large memory footprint (256 Mbytes or larger), within high-speed network segments, on processors that are readily upgradeable (and have been upgraded twice), are developed with modern software engineering tools, and are deployable to a variety of processors and operating system architectures.

The architecture exists to allow any programmer to support an accelerator device with access to CLIB, the console library, the relational databases, and without requiring access and expertise in microprocessor hardware and software.

Features

The architecture defines a default support and behavior for the Open Access Client (OAC) and its devices. The OAC programmer overloads or tunes behavior to the needs of the device.

By default, clients and their devices receive the following support. Devices support the reading, setting, basic status, basic control, analog alarm block, and digital alarm block properties. The Tevatron clock, fast time plot, snapshot plot, and alarm protocols are supported. Settings, status, and alarm blocks are downloaded from the device database. Settings are reflected into readings, basic control is reflected into basic status, and settings and status are uploaded to the database. Alarms are monitored by repository observation, then at one Hertz when going bad or going good. Digital alarms support status of status alarm updates. Fast time plot and snapshot plot rates of 60 Hertz are supported. Snapshot clock arming triggers, device arming triggers, external arming triggers, negative arming delays, clock sample triggers, and external sample triggers are supported. New database entries are downloaded, and up time and guaranteed read and write devices are supported. Client log files, device redirection and automatic test client instantiation are supported.

Definition

Open Access Front-end Clients are defined by application page D92,

Getting started

The following steps are required to fully implement a Java Open Access Client or Model.

Use ACNET page D92 to add a client to the database tables. Use the lowest class number available. Initially, set the OAS and OAC node and the OAS test node to your local engine.

Modify the MECCA service ACNET_STATUS by adding/modifying an error code in the file OAFP.TXT based upon the client class for reporting the unavailability of the client. (mecca/copy acnet_status; edit oafp.txt; mecca acnet_status)

DABBEL a heartbeat device for the client of the form G:OACnnn where nnn is the class in decimal with leading zeros. The node is the OAS node. The reading SSDN of the heartbeat device is 00nn/A5A5/A5A5/A5A5 where nn is the class in hexadecimal. Other devices for this client may be added where the SSDN should be all zeros except for the low order byte containing the class in hexadecimal. The EMC property is not required for the support of alarms.

Add this client to the ACNET node tables as a pseudo node.

DABBEL an alarm device for the client of the form J:node where node is the ACNET pseudo node name. Copy and modify from an existing J:node.

DABBEL a test device of the form Z:node where node is the ACNET pseudo node name. Copy and modify from an existing J:node, changing the node, description, and SSDN.

Create a project directory under the path j:\gov\fnal\controls\daq\oac\clients where the directory name is the client name. Reflection is used to find and load the classes for your client. Copy a trivial OpenAccessClient such as CACHE.JAVA, rename all the instances of "cache" to your client name, create a project and add this file, and build the project.

Without adding or modifying any additional code, your client will support all the ACNET data acquisition protocols. Downloads will populate the devices from the settings' database, settings will be reflected into readings, control into status, alarms will be monitored at 1 Hertz, and fast time plot and snapshot will be supported at 15 Hertz or greater.

Start a local engine. Your client will be instantiated as you begin testing by using the data source TestSource or redirecting ACNET applications using page D128.

Controlling the service and behavior of the client is available through the client's super class, the OpenAccessClientStub. The client may register as an observer to device instantiations, readings, settings, downloads, alarms and changes to the client's data repository.

After some development time, the steps required to make the client operational include setting the OAS node on page D92 to an operational engine, modifying database entries to point to the OAS node, modifying the ACNET node tables to reflect the operational change, and modifying the ACNET_STATUS unavailability error code if needed

Getting Involved

At this point, the OAC programmer is supporting devices in the same manner as CACHE, the OAC whose devices simply reflect settings into readings. To impress a different behavior on a device, the programmer may intercept readings and settings, control error returns, deposit values into the repository, specify plot class codes, define external plot triggers, and a myriad of other controls.

Generally, adding an observer to front-end activities and overloading functionality are the common means to change the default behavior of the OAC. Subclasses of ClientDataObserver offer the client opportunity to view access to and interact with client data during instantiation, read, and write operations. Observer opportunities include:

- AfterReadObserver
- AfterSetObserver
- AfterMinuteReadObserver
- AfterUserSettingObserver
- BeforePostingAlarmObserver
- BeforeReadObserver
- BeforeSetObserver
- BeforeUserSettingObserver
- DataConstructorObserver
- AfterDatabaseEditObserver
- PoolCancelObserver
- PoolUpdateObserver
- DaqPoolObserverInterface

Attaching a BeforeReadObserver to the reading property of the heartbeat device supports the heartbeat reading. Each read operation triggers the observer who calculates up time and updates the data repository before the repository is read to satisfy a data acquisition return.

Each client extends OpenAccessClientStub and may overload the stub's default functionality. Though it may not be apparent how each method might be overloaded, the methods within OpenAccessClientStub include:

```
public OpenAccessClientLog clientOpenLogFile()
public OpenAccessClientLog getClientLogFile()
public ReportCallback getClientReportTo()
public void setClientReportTo(ReportCallback reports)
public void setClientDownloadSource(DataSource source)
public DataSource getClientDownloadSource()
public void setClientDownloadDataItem(DataItem item)
public OpenAccessClientAlarmMonitor getClientAlarmMonitor()
public int getClientFastTimePlotClassCode(OpenAccessClientData data)
public int getClientSnapShotPlotClassCode(OpenAccessClientData data)
public StateTransitionTrigger getStateTransitionTrigger(int armSourceModifier)
public DataItem getClientDownloadDataItem()
```

public void sdaClientLog(WhatDaq device, int error, int offset, byte[] data, Date armTime, Date sampleTime)
public boolean isContinuousDownload()
public void setClientDescription(String description)
public String getClientDescription()
public boolean doSettingReflection()
public void setClientRepository(OpenAccessClientDataRepository repository)
public OpenAccessClientDataRepository getClientRepository()
public DataLoggerClientLoggingDisposition getClientLogger()
public boolean setClientLogger(String name)
public void clientDownloadComplete(boolean isDatabaseDownload)
public void clientActivation()
public void clientTermination()
public String clientStatistics()
public static String reportStatistics()
public final void engineShutdown(String reason)
public static void setClassBugs(boolean onOff)
public static boolean isClassBugs()
public String toString()

Existing OAC functionality

Several clients exist and should be utilized or modified to support devices appropriate to their charge instead of creating a new client.

BIGSAV and SCHSAV manages shot-setup, time-scheduled and operator initiated big saves. code

BLMLOG reads Booster beam loss monitor data as a large array device with readings a milliseconds apart over across several Booster cycle types, decomposes into OAC devices, and data logs the scalar devices. code

CACHE supports devices in RAM by accepting all the default behavior of an OAC.

CBSHOT, ESTORE, and PBSHOT are SDA companion clients. They receive scaled data from SDA and trigger client logging to the data loggers CShot, EStor, and PShot respectively. Each OAC supports 10 devices reflecting its status; these devices are set by the SequencedScheduler class who uses a device prefix from each OAC to form device names for the OAC's current store number (x:FILE), case (x:CASE), case arm time (x:ARMTIM), set (x:SET), set arm time (x:SETTIM), case disarm time (x:DISTIM), case end time (x:ENDTIM), snaps complete time (x:SNPTIM), and scalars complete time (x:DAQTIM). When a sequencer increments the shot or store number, SDA reinitializes and indicates its readiness by setting x:FILE to the shot or store number; i.e. SDA is ready when x:FILE == x:STORE. The test OAC auto test device (x:TSTSDA) may be used to start an automatic SDA run when ON and the test OAC is started or when turned ON when the test OAC is already running. Time is represented as the number of seconds since the SDA file was opened. The prefixes for CBSHOT, PBSHOT, and ESTORE are "C", "A", and "E" respectively.

CONSAR (for CONSolidation ARray) is a general purpose OAC which "consolidates", or concatenates, the readings from a set of devices into one big array of data. For each of CONSAR's devices, CONSAR looks in the database for the sub-devices whose readings are assembled into the array readout.

CRYTST front-ends a Moore APAC industrial controls system importing channels into the ACNET control system. Parts of CRYTST utilize a DLL and C++ code requiring a traversal of the JNI (Java Native Interface) boundary.

CSLDTR is an example of a front-end consolidator.

EVENTS data logs all software state transitions. EVENTS data logs Tevatron clock events and tracks the time since each event was last seen. This OAC supports devices of the form G:SNCExx where xx represents a clock event from 00 to FF. The devices are scaled for readback to represent the number of seconds since the event was last detected. The integer raw data represents milliseconds since last seen. This OAC also supports devices of the form G:ExxSCT where xx represents a clock event from 00 to FF. The integer raw data represents microseconds into the supercycle where the event was seen. This OAC also supports devices of the form G:ExxNUM where xx represents a clock event from 00 to FF. The integer raw data represents the count of events in the last multicast update containing this event. This OAC data logs all G:ExxSCT and every G:ExxNUM changes from the value one. This OAC also supports devices of the form G:ExxSUM where xx represents a clock event from 00 to FF. The integer raw data represents the sum of events in the current supercycle containing this event. This OAC data logs all G:ExxSUM at each supercycle reset. This OAC also supports the device of the form G:E00ERR which is a measurement of the timing error as the absolute difference of time reported by the UCD multicast and the engine's time of day. This OAC data logs G:E00ERR at each supercycle reset. Note, most data is stuffed into the repository by a BeforeRead observer. If alarm blocks are defined for these channels, code will have to be added to populate the repository periodically for channels with alarm blocks. code

EXPURT is the replacement for the VMS EXPORT OAC. It exports ACNET data using database configuration tables and udp.

FRCONS is the cryogenic refrigerator system's consolidator.

IIOP is a test OAC demonstrating the use of CORBA(Common Object Request Broker Architecture).

IPADC front-ends WebDaq100 analog to digital converter modules.

JNITST and JPLOT are development OACs for testing the JNI (Java Native Interface) and data logging.

LJALRM monitors data loggers and posts alarms on loggers that do not respond to pings.

MACALC (MAth CALCulation) computes readings for its devices from an algebraic expression involving other ACNET devices and arithmetic and logical operators. MACALC retrieves the expression for calculating each device from the database, making it very easy to add devices or change the calculations. MACALC's initial task was to replace most of the functionality of the VMS client APCALC.

MCRVCR manages devices for the MCR Video Server. Supported devices include G:VDR - when ON, connects to mcrops.fnal.gov; permits recorder control and when OFF, disconnects from mcrops.fnal.gov; permits no NEW recorder control. The devices G:VDR1, G:VDR2, G:VDR3, and G:VDR4 - when ON, permits recorder control and when OFF, permits no NEW recorder control. Control commands come from SDA, configured as snapshot plot requests where the sample rate is the frame rate. Maximum number of frames (without creating a new snapshot plot class) is 4096. Testing: MCRVCR receives its commands for video recording via an ACNET message. That message is normally composed and sent by a SequencedScheduler when arming a SDA case. The message includes a Boolean indicating if the SDA collection is operational or test. Testing involves ensuring this boolean is on, redirecting DAQ and instantiating a test MCRVCR, redirecting and instantiating a test SDA, firing the test state transition device for the case and set to collect a G:VDRn device.

MIRROR and MJRROR are identical Open Access Model clients supporting data acquisition redirection used for testing settings. Settings are reflected into readings.

MONITR reports alarms on monitored services, front-ends, Open Access Front-end clients, and engines that do not respond to pings. MONITR observes software state transition events after processing an initialization node status report job to announce nodes up and down. MONITR supports devices of the form J:node where node is the ACNET node or pseudo node name.

REMOTE allows remote systems to set ACNET devices by listening to a socket, receiving udp messages, and setting internal RAM devices. DZero and CDF provide data to the control system that then may be displayed on parameter pages, data logged, read by SDA, plotted, and the like.

SCHSAV is the scheduled big save OAC. It starts big saves four times a day.

SETSVR is an incomplete OAC targeted to log control system setting information for accountability.

TEVMTN manages the Tevatron mountain range display.

TIMEAV (TIME AVeraging) implements calculation of its devices' readings based on time averaging another device's readings over a specified period.
More that just averaging is implemented --- you can also simply sum up the sampled device's readings, or take the difference between subsequent averaging periods. A string specifying the averaging to implement is read from the database, making it very simple to add new devices and to specify how they are calculated.

Abstract Client Classes

The OACs named CONSAR, FRCONS, MACALC, and TIMEAV are all built from abstract client classes. The actual OAC class (e.g. TIMEAV) contains relatively little Java code. The design is that you can make additional instances of the abstract class with slightly minor variations, or to support a different set of devices for a different application.

ACNET

Introduction

ACNET is a connectionless peer-to-peer proprietary Fermilab accelerator binary networking protocol. Created about 1980, it remains the transport mechanism for data acquisition and much of the client, server communication. Though looking forward to an expansion of portable protocols such as RMI and CORBA, the data acquisition engines use ACNET extensively; the clients do not.

ACNET imposes a small header on an otherwise structure-less message. Message sizes are small. Request/no-reply, request/single-reply, and request/multiple-reply messaging is supported on a known socket using udp. A server thread is required to accept connection requests and direct incoming messages to tasks identified by a DEC Radix-50 name within the header.

Initialization

EngineStartup in the consolidate package initializes ACNET. Information about all ACNET nodes is read from the database table maintained by application page D98 and saved in a hash table. Another database table containing ip name for each node is initialized. An important difference between Java ACNET and other ACNET implementations is that Java ACNET will use (and requires) name servers to find and translate ip name to address. ACNET connections are made for threads with task names BOUNCE and ACNAUX. The former supports bandwidth testing, and the latter supports statistics returns to applications like page D38.

A socket is created for ACNET reads and writes. An ACNET read thread is created and maintains a blocking read on the ACNET socket. The read thread processes incoming messages for dispatching to connected task request and reply queues.

Most engine startups result in AcnetKiller messages sent to all ACNET nodes to cancel any outstanding multiple reply requests within the outlying node with infrequent replies that cannot be cleaned up otherwise.

Using ACNET

Once a named AcnetConnection is created, the connection can create AcnetSendRequest objects, transmit them to other task/node combinations, receive replies, cancel requests, and perform other operations on the connection. A connection may optionally declare itself to be an ACNET replier. Requests and replies are delivered within ACNET managed threads.

The ACNET implementation in Java supports multiple connections, deep resources, long message sizes, transmission at a message offset, timeouts for first replies and multiple replies. The implementation does not impose request and reply ids, delivers requests, replies, unsolicited special messages, and cancels to unique signatures, and all of its message deliveries are asynchronous.

To make ACNET dumps more readable and useful, the AcnetReply interface forces users to implement:

```
public boolean isRequestValid(AcnetConnection. AcnetSendRequest request)
public String requestDescription();
```

The AcnetMessage object support the composition and decomposition of ACNET messages with get and put operations for most data types.

Connected tasks

ACNAUX supports the retrieval of ACNET statistics, see page D38.
ALARMR is the Open Access Front-end alarm server.
BOUNCE supports ACNET bandwidth testing.
BPMVAX receives scaled BPM values from LBOE on CNS40 for SDA.
BRIDGE supports VMS access to Java data including node status.
BUGS supports ACNET debugging message requests.
CACHEX deposits raw BPM data from SDA to reflection devices in the OAC CACHE for LBOE to read, scale and transmit to BPMVAX.
ERRSVR receives ACNET error and bandwidth statistics from data acquisition clients.
FSMRPT transmits software state transition device setting requests to FSMSET, the states front-end.
FTPMAN is the Open Access Front-end connection for plot requests.
FTPPLT supports the fast time plot pool making requests of the front-end FTPMAN task.
KILLER is the ACNET initialization task that sends messages to clean up previous ACNET traffic.
LJPING sends ping (heartbeat) messages to data loggers.

LJSTAT sends Save/Restore one shot pool statistics client logging messages to the STATES data logger.
LMBRJK supports data logger retrieval protocol requests.
LOGGET issues data logger retrieval protocol requests.
MCRVCR is an Open Access client supporting video recording requests within Sequenced Data Acquisition.
PINGER sends ping (heartbeat) requests to POOLER.
POOL sends requests to front-end RETDAT and SETDAT tasks. Each ACNET connection has one reply thread. Pool connects as multiple users to get several reply threads. The connected tasks and uses are:

- POOLR, repetitive
- POOLP, ping
- POOL1, one-shot
- POOLH, hard event
- POOLS, settings
- POOLE, soft events
- POOLX, other

POOLER supports front-end consolidation requests from other engines.
RETDAT is the Open Access Front-end connection for reading requests.
SETDAT is the Open Access Front-end connection for setting requests.
SEQSCH is a SequencedScheduler for SDA; sends video recording requests to MCRVCR.
SETSVR receives settings information for accountability logging.
SNAP requests plot class codes from front-ends.
UPLOAD is the Open Access Front-end database setting upload task.

Appendix

History

Impetus

Languages, operating systems, platforms

Object-oriented languages are preferred over the function-oriented languages C and Fortran. Digital Equipment Corporation's VAX/VMS is an obsolete operating system. The central and console platforms cannot be upgraded beyond the MicroVAX 3100 M90, a system that is almost 10 years old.

Recruitment and retention

It is difficult to recruit and retain programmers for a shop perceived to be using obsolete languages, operating systems, and platforms.

Software engineering tools

Modern software engineering tools are not emerging on obsolete operating systems.

Criteria

Reusability

Replacing hundreds of man-years of application software necessitates a great deal of reuse. Data acquisition redirection has shown, for example, that a parameter page obviates the need for a Save/Restore display page, and one plot package should serve fast time plotting, snapshot plotting, and data logger plotting.

Commodity

High volume, inexpensive commodity products, particularly computers, should be exploited. The console and central system upgrade of ten years ago had a $1.7M budget, a level of funding difficult to obtain today.

Accessibility

The X-Window technology employed in the present system brought accelerator controls to the office and the home. Controls' upgrades should provide easy accessibility for diagnostics and operation of the accelerator and its systems anywhere on the planet.

Staged deployment

Front-ending old controls

The upgrade will encompass several years and an operating accelerator. New controls must coexist with old controls sometimes providing dual access to systems and services.

Central Services

Central Services, by definition, do not have a console graphical user interface and provides targets for a new data acquisition design and implementation.

Core applications

Save/Restore, data logging, plotting, and parameter page applications form a core of needs that new controls' data acquisition, storage, and retrieval target.

Maintenance retirement

The ultimate goal of new controls is to retire systems thereby reducing the maintenance burden. When this upgrade is successfully completed, all VAX/VMS machines will be gone, all C and Fortran software will be retired, all proprietary protocols having a portable protocol replacement will be retired, and any custom software with a commodity or public domain replacement equivalent will be gone.

Selection

Searching

X-Window replacement of CAMAC hardware was a significant element of the 1990 upgrade. Because X-Window programming was difficult and an upgrade goal was to

port most applications without change, a proprietary graphical user interface library with roots from the 1980 upgrade was ported. A post-upgrade goal was to offer an open X-Window or MOTIF value added package but was never fulfilled.
In the mid-1990's, integrated development environments, software engineering tools and fourth generational languages seemed to be poised to change the face of controls' computing. However, tools and vendors appeared and disappeared with alarming rapidity and most offered very proprietary solutions.
The emergence of Java was accompanied by a reduction in alternatives, and it seemed that Java would have to succeed or fail before other initiatives could gain momentum.

Engine and Java Links

Summer, 1997
X-Window console in Java

November, 1997
Web Utilization in Controls at Fermilab paper
Web Utilization in Controls at Fermilab presentation
ICALEPCS '97
Web Utilization in Controls at Fermilab - division presentation
Beams Division seminar

February 2, 1998
Java Data Acquisition Engine
Empowering Java and Data Acquisition
Proposal to department

July, 1998
BD Software Controls Workshop

July, 1999
Next Generation Accelerator Controls at Fermilab
White paper

January, 2000
Central Tier
Applications Design Review
Division review of controls

April, 2000
Java Services for the Collider Run
Beams Division seminar

Code Example, monitor Tevatron clock events 0x00 and 0x02 and state transition event V:PING

```
import gov.fnal.controls.daq.datasource.*;
import gov.fnal.controls.data.items.*;
import gov.fnal.controls.daq.events.*;

public class MonitorEvents implements DataEventObserver
{
      public MonitorEvents()
      {
            EventDataItem item = new EventDataItem ();
            item.add(new ClockEvent (0), this);
            item.add(new ClockEvent (2), this);
            item.add(new StateEvent ("V:PING",
                  StateEvent.FLAG_ALL_VALUES), this);
            item.startObserving(new AcceleratorSource (), 0);
      }

      /**
      * A requested event has arrived.
      * @param request original DataEvent request from the observer.
      * @param event reply to original DataEvent request; the DataEvent
      *        which just occurred that matched the 'request'.
      */
      public synchronized void update(DataEvent request,
                                        DataEvent reply)
      {
            System.out.println("Requested event " + request +
                  " has arrived: " + reply);
      }
}
```

Explanation

The first 3 lines import packages supporting data acquisition that are needed for the code that follows.

The next line defines the class and declares that it implements DataEventObserver interface which is satisfied by the update method.

The next line is the constructor for this class.

The next line creates an EventDataItem, a class that supports the retrieval of Tevatron clock events and state transition events.

The next lines add Tevatron clock events zero and two as well as the state transition device "V:PING" to the EventDataItem, and 'this' is the object that has implemented the DataEventObserver interface.

The next line starts observing events from the accelerator. In this example, that implies the Tevatron clock and the software state transition devices as opposed to some historical record of event transitions.

The update method satisfies the DataEventObserver interface and prints the requested and received event to terminal output.

Alternately, events may be observed by creating and starting a DaqJob.

Code Example, start a BigSave

```
import gov.fnal.controls.daq.acquire.*;
import gov.fnal.controls.daq.datasource.*;
import gov.fnal.controls.data.items.*;
import gov.fnal.controls.daq.events.*;


(within some class)

DaqUser user = new DaqUser ("BigSaveTry") ;

DataSource from = new AcceleratorSource();
DataSource to = new SavedDataDisposition ("SaveFile", 100, false);
DataItem  item = new DaqAllItem ();
DataEvent event = new OnceImmediateEvent();
DaqJobControl control = new DaqJobControl();

DaqJob job = new DaqJob(from, to, item, event, user,control);

try
{
      System.out.println("Starting a BigSave at " + new Date());
      job.start();
      job.waitForCompletion();
      System.out.println("Finished a BigSave at " + new Date());
}
catch (Exception e)
{
      System.out.println("whoops, job trouble caught: " + e);
}
```

Explanation

The first 4 lines import packages supporting data acquisition that are needed for the code that follows.

The next line establishes a connection with a server engine, in this case the default engine since it is not specified. The user may be prompted to log in so the engine can determine the privileges of this user. Most applications need but one DaqUser connection for all the DaqJob (s) of their application.

The next line specifies the data source is the accelerator, i.e. collect data from the front-ends in real time.

The next line specifies the disposition to be the save file number 100.

The next line specifies that all devices (as understood by the disposition) should be collected.

The next line creates a collection frequency object specifying a collection of but once.

The next line creates a default job control object.

The next line creates the job.

The next few lines write a start message, starts the job, waits for the job to complete, and writes a finished message. Catching and printing the exception to the terminal will describe a problem with the job.